Towards Efficient Location and Placement of Dynamic Replicas for Geo-Distributed Data Stores

Pierre Matri
Ontology Engineering Group
Universidad Politécnica de Madrid
Madrid, Spain
pmatri@fi.upm.es

Alexandru Costan
IRISA / INSA Rennes
Rennes, France
alexandru.costan@irisa.fr

Gabriel Antoniu
Inria Rennes
Bretagne-Atlantique
Rennes, France
gabriel.antoniu@inria.fr

Jesús Montes
Universidad Politécnica de Madrid
Madrid, Spain
jmontes@fi.upm.es

María S. Pérez
Ontology Engineering Group
Universidad Politécnica de Madrid
Madrid, Spain
mperez@fi.upm.es

ABSTRACT

Large-scale scientific experiments increasingly rely on geo-distributed clouds to serve relevant data to scientists worldwide with minimal latency. State-of-the-art caching systems often require the client to access the data through a caching proxy, or to contact a metadata server to locate the closest available copy of the desired data. Also, such caching systems are inconsistent with the design of distributed hash-table databases such as Dynamo, which focus on allowing clients to locate data independently. We argue there is a gap between existing state-of-the-art solutions and the needs of geographically distributed applications, which require fast access to popular objects while not degrading access latency for the rest of the data. In this paper, we introduce a probabilistic algorithm allowing the user to locate the closest copy of the data efficiently and independently with minimal overhead, allowing low-latency access to non-cached data. Also, we propose a network-efficient technique to identify the most popular data objects in the cluster and trigger their replication close to the clients. Experiments with a real-world data set show that these principles allow clients to locate the closest available copy of data with small memory footprint and low error-rate, thus improving read-latency for non-cached data and allowing hot data to be read locally.

1. INTRODUCTION

The rapid development of cloud computing solutions such as Amazon Web Services [2], Microsoft Azure [5] or Google Cloud Platform [3] has lead a number of applications to move from dedicated hardware to public clouds. This commonly allows the user to reserve and to use resources in multiple datacenter locations. Among these applications, scientific experiments tend to become geo-distributed as well. Such geo-distribution provides low-latency data access for scientists worldwide while minimizing bandwidth utilization and improving fault-tolerance as well as disaster-recovery. For instance, the MonALISA monitoring backend [20] of ALICE (A Large Ion Collider Experiment) [30] is distributed over 300 sites around the world. This raises the question of *data locality*, i.e. the location of the data relative to the computational resources.

Content Distribution Networks [25], or *CDNs*, located between the end-user and the origin data location, help reducing content access latency by caching it as close as possible to the end-user. However, being loosely coupled with the underlying data storage, they are usually only suitable for content that changes rarely, such as media files or static resources. Different replication strategies are needed to provide the same data locality properties to frequently updated content such as database objects.

Replicating all content at each site is a simple way to increase read performance. However, it can lead to significantly degraded read performance due to the additional synchronization needed. It also results in poor network and storage resource usage by unnecessarily replicating data to sites from which it is only rarely queried. To alleviate from this, an option is to statically choose one (or several) sites for replication when the data is first saved into the system. Although resource overhead will be significantly lower, and assuming a pertinent site choice, this still ignores the fact that data popularity can change over time, potentially resulting in a non-optimal data locality.

Using real-time metrics to dynamically decide when to create (or remove) additional replicas while retaining strong data consistency can be challenging. Many previous attempts [24] rely on one or multiple centralized metadata

servers that indicate to the clients if and where a specific piece of data can be read locally. These central servers can potentially become hot spots under highly concurrent reads, being placed on the critical path of any read or write. The added synchronization needed between metadata servers will further decrease the write performance of the system. This centralized approach is hardly compatible with the largely distributed DHT-based design of storage systems such as Dynamo [14], Cassandra [19] or Týr [21], making poor usage of their communication protocols between nodes. Since all of these systems rely on a gossip-based [15] failure detection algorithm and periodically exchange messages between nodes, we argue that it is desirable for caching algorithms to use these messages in order to disseminate live object usage information, consequently reducing the network overhead.

In this paper, we are introducing a set of algorithms and design principles that allow to *dynamically replicate popular data objects as close as possible to the end-users* and to consequently guarantee the lowest-possible request latency. Our approach leverages *completely decentralized metadata* in order to avoid traversing unnecessary network hops and to guarantee efficient location of all data objects. These features make our proposal particularly well-suited for loosely-coupled DHT- and gossip-based storage systems.

To the best of our knowledge, we are proposing the first decentralized, dynamic replication system that allows the clients to locate the closest copy of the data independently of any metadata server. Our contributions are these:

- **A probabilistic usage-aware dynamic replication module** running on each node of the cluster, which efficiently collects and disseminates usage metrics (Section 2.1). It independently orders additional copies of the data to be created or deleted based on this information. The underlying algorithms rely on probabilistic principles to keep the metadata overhead and performance cost as small as possible.

- **A probabilistic decentralized data-location algorithm** embedded in the cluster nodes and in client libraries, allowing users to efficiently locate additional data copies without having to contact any centralized metadata server (Section 2.2). This ultimately allows for low-latency queries, avoiding hot spots in the cluster and improving its horizontal scalability.

- **An experimental study of a proof-of-concept implementation** leveraging the above principles and showing that the dynamic data replication set forth by our proposal does not come at the cost of reduced performance or increased request latency (Section 3).

We briefly discuss the obtained results (Section 4) and review the related work (Section 5). Finally, we conclude by outlining future work on our proposal (Section 6).

2. DESIGN OF A SELF-ADAPTIVE DATA STORAGE SYSTEM

In this paper, we assume one cluster whose nodes are located in different geographical locations, or *sites*. Each node can talk directly to any other on using the network, and all nodes are equal in terms of responsibilities in the cluster.

State-of-the-art distributed hash table based storage systems such as Dynamo introduce the concept of *smart clients*, i.e. clients that use cluster state information to route requests to the appropriate node directly. We propose to extend these capacities with dynamic replication awareness. Instead of using a metadata server to locate a piece of data it wants to read, a client is able to independently determine the closest available location of the data. Clients supporting this level of functionality are called *smart clients*. In order to preserve backward compatibility with other clients, referred to as *naive clients*, they will continue working properly but will not benefit from an optimal location of the data objects.

2.1 Collection of usage metrics

2.1.1 Preferred site indication

We assume that each client maintains a list of every site in the cluster ordered by preference (such as average network latency, geographical proximity or any other relevant metric). When submitting a read request to any node in the cluster, smart clients include an additional piece of information: the *preferred site*. The preferred site is the site the client would like the piece of data being read to be available in. This site is usually the closest site from the client, or the site with the lowest read latency.This preferred site is embedded in the request even if it has been sent to a node belonging to it.

2.1.2 Per-node popularity metrics collection

In order to be able to suggest additional copies of objects to be created in the cluster, each node keeps track of the most popular couples $(object, preferredsite)$ from the requests it receives from clients during a read.

The identification of these couples is performed by adapting a lightweight streaming algorithm designed to find the elements with the highest frequency in a stream of data: Space-Saving [23]. We recall here a few of its key features that will be leveraged later. Its probabilistic nature allows to identify the most frequent elements while keeping tight error guarantees and using a small and predictable amount of memory. The output of the algorithm is a list of candidate frequent elements, with multiple associated counters:

- c_e is the counter structure for an element e,
- $c_e.\hat{h}$ the hit counter for this element, i.e. the total estimated number of hits of the element e,
- $c_e.\varepsilon$ is the error counter for the element e, i.e. the error margin of the hit counter for this element.

The basic Space-Saving algorithm is detailed in the pseudocode of Algorithm 1.

The Space-Saving algorithm identifies the k most frequent elements in a stream S. We are more interested in the most recent frequent items than in the historic frequent items. We then need to reinitialize the Space-Saving summary frequently, using only one configurable time window. In order to avoid starting each of these periods with an empty summary and because the Space-Saving structure has been demonstrated to be mergeable [9], we merge the Space-Saving summary from the current time window t, C_t, with the one from the previous time window C_{t-1}. As such, the frequent element summary for the time window t, F_t, is given by $F_t = \text{MERGESPACESAVING}(C_t, C_{t-1})$.

Algorithm 1 Basic *Space-Saving* algorithm

Input: $k > 0$ counters, element stream S
Output: summary containing most frequent elements and their estimated frequency, C

```
procedure SpaceSaving(S, k)
    C ← InitializeCounters(k)
    for all elements e in S do
        if e is monitored then
            let c_e be the counter structure for e
            c_e.ĥ ← c_e.ĥ + 1
        else
            let e_m be the element with least hits, c_em.ĥ
            replace e_m with e
            c_e.ε ← c_em.ĥ
            c_e.ĥ ← c_em.ĥ + 1
        end if
    end for
end procedure
```

2.1.3 Cluster-wide popularity metrics dissemination

Our algorithm builds on a gossip infection-style, weakly-consistent protocol [15, 13] to disseminate object popularity information across the cluster with the lowest possible network overhead. Each node periodically sends to other randomly selected nodes in the cluster its own Space-Saving summary for the current time window, tagging it with the local time. In order to quickly relay the information cluster-wide, the nodes piggyback these messages with recently-received information from other nodes, including their associated time. Each node keeps a view of the most popular items for each node in the cluster, updating its state from another node only if the received one is more recent (based on its timestamp value). As such, for a n-node cluster and a configured value of k counters per summary, the size of the complete structure is predictable and in the order of $O(kn)$.

2.1.4 Additional replica placement

Our proposal for replica placement relies on consistent hashing [17], as implemented by various decentralized storage systems such as Chord [29], Dynamo [14] or Týr [21] for data placement. Given a hash function $h(x)$, the output range $[h_{min}, h_{max}]$ of the function is treated as a circular space (h_{min} sticking around to h_{max}). Every node is assigned a different random value within this range, which represents its position on the ring. The node responsible for an object *obj* with key k is determined by the result of $h(k)$, giving a unique position h_k on the ring. The first node encountered while walking the ring past this position is called the *coordinator node* of *obj*. The nodes to store additional replicas of an element are obtained by continuing walking the ring passed the coordinator node until an appropriate number of nodes are found. All these nodes are called *natural storage nodes* of *obj*.

Using the same ring, the node that will be chosen to hold an additional replica of the object *obj* in a site s, if any, is the first node in the site s encountered while continuing walking the ring past the last natural storage node for *obj*. This node being chosen deterministically, any additional replica for a given object on a specific site will always be stored on the same node.

2.1.5 Dynamic replica creation

Merging all the collected summaries received from every node gives each node an overview of the most frequent couples $(object, site)$ cluster-wide. This merge is performed using the same Space-Saving merging function used in Section 2.1.2. Each node checks periodically whether such couples for objects it is the coordinator node for appear in this cluster-wide summary. If so, for each site at which an object *obj* is popular, a new replica is created at that site, if none exists already. The specific node on which such replica is created is determined as explained in Section 2.1.4.

Replica creations are advertised to all other nodes in the cluster using the gossip protocol defined in Section 2.1.3. Each node keeps a list of all objects for which additional replicas have been created, as well as the sites they have been replicated on.

2.1.6 Dynamic replica removal

Whenever the popularity of an object drops, additional replicas dynamically created are no longer needed and must be deleted. The node holding an additional replica for an object *obj* is responsible of independently deleting that copy in order to make room for other replicas. This deletion happens after an object previously present in the cluster-wide summary disappears from it.

In order to prevent replicas being repeatedly created and deleted for objects with popularity fluctuations near the Space-Saving threshold, this deletion may be delayed after a grace period greater than the summary window length defined in Section 2.1.2. Similarly to replica creation, this deletion is advertised using the same gossip protocol.

2.2 Accessing dynamic replicas

2.2.1 Decentralized data location

When a client reads an object, it tries to locate the closest copy of a desired data object within the cluster. To do this, it needs sufficient information in a memory- and network-efficient fashion. Bloom filters [10] provide an ideal base for this. Their compact nature makes them easy to forward over the network and lightweight to store in the clients memory, while providing fast lookup and tight guarantees on false positive rates.

Since replica creation or removal is advertised cluster-wide using gossip, each node is able to provide this information to any client. As such, we allow smart clients to address a request to *any node on any site* and receive in response a Bloom filter summarizing the current additionally replicated objects throughout the cluster, identified by the couple $(object, site)$.

To read an object *obj*, a client needs the Bloom filter obtained from any node of the cluster as well the ordered list S of preferred sites. The site s to send a request to is either the first one in the S list or the one for which the Bloom filter search for (obj, s) returns a positive result. The algorithm is detailed in the pseudocode of Algorithm 2.

At the chosen site s, the request is addressed to the node that would hold a dynamic replica, using the same deterministic algorithm described in Section 2.1.4.

2.2.2 Server read request handling

When a node receives a request for an object it holds, it responds to the client normally. In some cases however, our

Table 1: Simulated round-trip latencies between sites (ms)

	US East	US West	Europe	Asia	Pacific
US East	0.25	35	70	145	185
US West	35	0.25	105	110	150
Europe	70	105	0.25	120	310
Asia	145	110	120	0.25	140
Pacific	185	150	310	140	0.25

Algorithm 2 Object location algorithm

Input: object to read obj, list S of all sites ordered by preference.

```
ss ← ∅

function LocateClosestReplica(obj)
    if ss = ∅ then
        let s_r be a random node in the cluster
        ss = RequestBloomFilter(s_r)
    end if
    let N be the list of natural sites for obj
    for all site s in S do
        if s in N or (obj, s) in ss then
            return s
        end if
    end for
end function
```

algorithm can cause the client to contact a node not holding the desired data. This can happen either because the client used an outdated Bloom filter (i.e. the filter returned a false positive because the cluster had not yet converged on the current replication status when the Bloom filter was obtained) or because the client was of naive type. Should the client contact the wrong node in such case, that node will directly forward the request to the closest node holding the data according to its knowledge of the replicated objects in the cluster, or using the DHT information.

Relaying the request instead of responding to the client incurs fewer messages, making a false guess from the client result in only a single message overhead compared to a successful guess, furthermore addressed to a close site.

2.2.3 *Bloom filter refresh*

As replicated objects may change depending on the performed user requests, the Bloom filters need to be periodically refreshed to keep the false positive rate of our location algorithm as low as possible. As such, after a configurable period of time, clients will contact a random node of the cluster to request an up-to-date version of the summary.

To cope with cases where the configured summary validity period is too large for some usage patterns, the client keeps a count of false positives or false negatives. Should the client address the request to a server not holding the desired piece of data, the server relays the query as usual to the correct node but flags the answer as erroneous. This may happen for instance because it has been deleted after the client bloom filter has been last updated. We call this case a *false positive*. Inversely, if the client accesses a remote object replica and a closer one existed, which we call a *false negative*, the server responding to the client flags the response message as well. A counter of wrong assumptions is maintained by each client, and set to 0 every time the summary is updated. For each false positive or negative, the client increments its counter. It updates its bloom filter as soon as the counter reaches a configurable threshold.

2.2.4 *Additional optimizations*

Since the Bloom filter summary of the additional replicas can be requested from any node in the cluster, clients can piggyback update requests to any request to the cluster. In turn, nodes are able to piggyback the latest summary to the response. This results in a sensible reduction of the network utilization.

Additionally, a desirable behavior of the location algorithm is to allow nodes to use newly-created replicas of popular data as soon as possible after they have been created, without waiting for a summary refresh by clients. As such, whenever a cluster node receives a read request from a client, it can check if another copy of the data is available closer to the client. In that case, the node piggybacks this information to the response, along with an updated summary that will allow the client to read from the closest available copy in the future.

3. EVALUATION

3.1 Experimental setup

We evaluated our design using a synthetic simulation on 80 processes running on a single machine, replicating the behavior of multiple nodes over multiple sites – 5 sites composed of 16 machines each. Gossip between processes is based on the SWIM protocol [13], implemented by Serf [6]. Latency between nodes is simulated using the native Linux Traffic Control [4] utility. Latency settings have been set to realistic values measured by Verizon [7] on their own network, and are detailed in Table 1. The simulation itself was implemented using approximately 700 lines of Python code, not counting existing open-source implementations of Bloom filters and Space-Saving algorithm.

Object requests were simulated using an open data set composed of real usage data collected at the University of Indiana [22]. It is composed of the URLs of more than 25 billion user requests collected over two years. Usual web access data have been demonstrated to follow a Zipfian distribution [26, 8], causing hot spots on storage clusters. In our experiments, we consider URLs to be a data object and simulate read requests originating from all 5 different, geographically disperse sites.

For our experiments, the Space-Saving time window length defined in Section 2.1.2 was set to 30 seconds, the number of counters to 128, and the removal grace period defined in

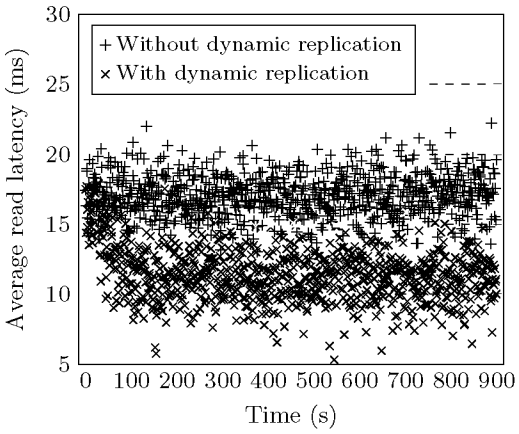

Figure 1: Achieved read latency with and without dynamic replication enabled.

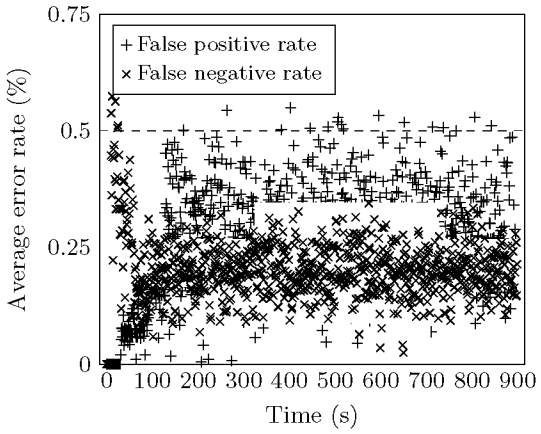

Figure 2: Measured false positive and false negative rates.

Section 2.1.6 to 90 seconds. This experiment was ran for a total duration of 15 minutes.

3.2 Object replication

We argue that our technique is able to efficiently identify the most popular data objects throughout the cluster and replicate them as close as possible to the interested clients. Improper identification of these popular objects would result in no significant decrease in the average read latency. In that case, non-frequent data objects would be replicated, and these would be unlikely to cause any noticeable difference in the obtained results.

In our results, plotted in Figure 1, we can observe a clear decrease of the mean request latency with dynamic replication enabled compared to the results without replication. We notice at the beginning on the curve the initial period of time in which the popular data objects are being identified and progressively replicated. High variability in the results is caused by the inner characteristics of the data set, where frequent items tend sometimes to be grouped

With our dynamic replication technique, after result stabilization, the achieved average read latency dropped by 38%, from 16 ms to 10 ms. It is also interesting to observe that that the frequent elements were identified quickly, in under three times the Space-Saving window length. The number of replicated objects stabilized at an average of 147 replicated objects in the cluster, slightly more than the configured number of counters in the Space-Saving algorithm, due to the grace period before dynamically created replicas of previously-popular objects are deleted.

3.3 Object location

Our client-side algorithm is able to locate the additional replicas in the cluster with minimal error rate, thanks to the fast convergence of the gossip information dissemination and to the guaranteed error rate of the Bloom filter summary sent to the clients. We want to evaluate the accuracy of our proposal, by measuring and plotting the error rates of this algorithm. We separate the *false positives* – when a node is wrongly believed to hold a replica of the object, from the *false negatives* – when the request is sent to a non-optimal replica of the object, i.e. when a closer replica existed. A false positive indicates that a client requested an object replica at a site where it was not available. This can happen if the bloom filter of the client is outdated and the replica has been deleted. A false negative indicates that the client has requested an origin copy of the data while a local replica existed. This may occur if the bloom filter of the client was last updated before this replica was created.

The results are depicted in Figure 2. We observe that our algorithm correctly identifies the most-optimal replica of the desired data in more than 98% of the cases, independently and without requiring the assistance of a metadata server.

The initial high false negative rate is caused by the first replicas being created, and is explained by the short period of time before the clients are made aware of these copies. The error counter described in Section 2.2.3 helps to quickly reduce this error rate.

Not surprisingly, the false positive rate increases around 120 seconds after the start of the experiment. This is caused by the first additional replicas being deleted shortly after the configured 90-second grace period plus 30 seconds Bloom filter window length.

4. DISCUSSION

4.1 Impact of the workload type

Our work targets applications in which reads dominate over writes. For applications with a lower read-to-write ratio, the potential gains of dynamic replication might not be as clear. Specifically, should an object be updated, dynamic replicas need to be kept synchronized with the original data. As the number of dynamic replicas of any given object grows, the overhead of this synchronization grows as well. This will likely result in a decreased write performance for this object, which might hinder the performance of the application.

Future work will focus on experimenting our algorithm on top of a production system to evaluate the actual impact of our proposal on the performance of a real-world application with different workloads.

4.2 Decentralized message dissemination

Our proposal relies heavily on a gossip protocol being used by the underlying storage system. For storage systems not based on a gossip protocol, such as HDFS [27], the benefits

of adding such a communication layer instead of relying on the existing centralized architecture still has to be evaluated.

5. RELATED WORK

Self-adaptive storage. These systems emerged in the context of autonomic computing, a paradigm inspired by biology (i.e. by how the human nervous system reacts to external changes in an autonomous manner, through unconscious reflexes in order to adapt our body to its needs and to the environment without requiring our attention). IBM has introduced a reference model for autonomic control loops [1], based on monitoring, analysis and reactions, currently used by most autonomic storage systems. In [11] Carpen et al. make a first step towards enhancing with self-adaptive support a massively distributed storage system. They add an introspection layer that collects data about the usage of storage resources and data access patterns relying on the MonALISA [20] monitoring framework. However, in contrast to our solution, this approach lacks the reaction phase that closes the control loop by adjusting the storage system according to the tracked data. More recent solutions turn to adapting some specific parameters, e.g. the replication factor [18]. Here, a DHT-based self-adapting replication protocol is introduced to determine the locations of replicas and then to autonomously adjust their number to deliver a configured data availability guarantee. However, these works focus on the relationship between the number of replicas and performance, completely ignoring the impact of replica location on query efficiency. D-Tunes [24], the related work most similar to ours, tries to fill this gap. It is based on some self-tuning algorithms that can adapt to workload changes over short time-scales by automatically configuring parameters like replication factors, consistency levels and readjusting read/write priorities while judiciously recommending data placements over longer time-scales. However, D-Tunes does change replication settings on a global-scale, making it difficult to perform fine-grained replication optimization on a per-object basis.

Dynamic replication. A vast number of works acknowledge that static, manual replication introduces a serious overhead and can drastically affect the storage performance. This is the case with geo-replicated datastores such as Spanner [12] and Cassandra [19], in which replication strategies are manually configured by the application developers. An alternative consists of dynamically distributing workloads by replicating and migrating replicas among storage nodes. Since dynamic multiple-location replication is NP-complete in nature, most of the existing strategies for dynamic replication are typically based on so-called single-location algorithms for identifying a single site for data replication. Dong et al. [16] transform the multiple-location problem into several classical mathematical problems with different parameter settings, for which efficient approximation algorithms exist. However, they don't consider the impact of replication granularity on performance and scalability. Wei et al. [31] address this issue by trying to answer the question: how many replicas the system should keep at least to satisfy availability requirements? To this end, a model is developed to express availability as a function of replica number. This approach, however, only works within a single site, as it assumes uniform bandwidth and latency, which is not the case with the geo-distributed workloads that we target. Inspired by the P2P systems, [28] proposes an adaptive decentralized file replication algorithm that achieves high query efficiency and high replica utilization at a significantly low cost. The idea is to select query traffic hubs and frequent requesters as replica nodes, and to dynamically adapt to nonuniform and time-varying file popularity and node interest. Unlike current methods and similarly to our approach, they create and delete replicas in a decentralized self-adaptive manner guaranteeing high replica utilisation. While this solution works gracefully in a P2P environment, its assumption of a known underlying network topology (which does not hold in a cloud environment) and the use of additional intermediate nodes (hubs) along the path between the client and the servers make it unusable in modern storage systems.

6. CONCLUSIONS

In this paper, introduce techniques and algorithms designed to help reduce read latency in a geographically distributed storage cluster. To do so, we propose a method to collect live request metrics using probabilistic algorithms in order to identify the most popular data objects. These metrics are disseminated throughout the cluster using gossiping. We further propose another probabilistic, decentralized algorithm for efficiently locating the closest available copy of the data. It does not require the use of a centralized metadata center, thus improving the fault-tolerance and horizontal-scalability of the cluster, while at the same time reducing the client read latency.

Our experiments over a large real-world data set show that these techniques can efficiently identify and duplicate the most frequently requested data objects in a cluster. Additionally, they allow the client to independently and probabilistically locate the additional copies of the data with a low error rate.

In order to demonstrate the performance of our approach, future work will focus on implementing these techniques on top of a real storage system such as Cassandra. We also plan to evaluate its performance with additional real-world use cases and data sets. Finally, we are investigating the use of machine-learning techniques to be able to identify with higher accuracy the most important data objects to be replicated.

7. ACKNOWLEDGMENTS

This work is part of the BigStorage project, funded by the European Union under the Marie Skłodowska-Curie Actions (H2020-MSCA-ITN-2014-642963). The authors would like to thank the reviewers who provided valuable counsel and expertise that greatly helped this research.

8. REFERENCES

[1] An architectural blueprint for autonomic computing. Technical report, IBM, June 2005.

[2] Amazon Web Services. https://aws.amazon.com/, 2016. [Online; accessed Feb-1016].

[3] Google Cloud. https://cloud.google.com/, 2016. [Online; accessed Feb-1016].

[4] Linux Traffic Control. http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html, 2016. [Online; accessed Mar-1016].

[5] Microsoft Azure. https://azure.microsoft.com/en-us/, 2016. [Online; accessed Feb-1016].

[6] Serf by HashiCorp. https://www.serfdom.io/, 2016. [Online; accessed Mar-1016].

[7] Verizon Enterprise Solutions IP Latency Statistics. http://www.verizonenterprise.com/about/network/latency/, 2016. [Online; accessed March-1016].

[8] L. A. Adamic and B. A. Huberman. Zipf's law and the internet. *Glottometrics*, 3(1):143–150, 2002.

[9] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi. Mergeable summaries. *ACM Transactions on Database Systems (TODS)*, 38(4):26, 2013.

[10] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[11] A. Carpen-Amarie, A. Costan, J. Cai, G. Antoniu, and L. Bougé. Bringing Introspection into BlobSeer: Towards a Self-Adaptive Distributed Data Management System. *International Journal of Applied Mathematics & Computer Science*, 21(2):229–242, 2011. To appear.

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, Aug. 2013.

[13] A. Das, I. Gupta, and A. Motivala. SWIM: scalable weakly-consistent infection-style process group membership protocol. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 303–312, 2002.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[15] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, et al. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[16] X. Dong, J. Li, Z. Wu, D. Zhang, and J. Xu. On dynamic replication strategies in data service grids. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 155–161, May 2008.

[17] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[18] P. Knežević, A. Wombacher, and T. Risse. DHT-Based Self-adapting Replication Protocol for Achieving High Data Availability. In E. Damiani, K. Yetongnon, R. Chbeir, and A. Dipanda, editors, *Advanced Internet Based Systems and Applications*, pages 201–210. Springer-Verlag, Berlin, Heidelberg, 2009.

[19] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[20] I. Legrand, H. Newman, R. Voicu, C. Cirstoiu, C. Grigoras, et al. MonALISA: An agent based, dynamic service system to monitor, control and optimize distributed systems. *Computer Physics Communications*, 180(12):2472 – 2498, 2009.

[21] P. Matri, A. Costan, G. Antoniu, J. Montes, and M. Pérez. Týr: Efficient Transactional Storage for Data-Intensive Applications. Technical Report RT-0473, Inria Rennes Bretagne Atlantique ; Universidad Politécnica de Madrid, Jan. 2016.

[22] M. Meiss, F. Menczer, S. Fortunato, A. Flammini, and A. Vespignani. Ranking web sites with real user traffic. In *Proc. First ACM International Conference on Web Search and Data Mining (WSDM)*, pages 65–75, 2008.

[23] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory*, ICDT'05, pages 398–412, Berlin, Heidelberg, 2005. Springer-Verlag.

[24] S. P. N., A. Sivakumar, S. G. Rao, and M. Tawarmalani. D-tunes: self tuning datastores for geo-distributed interactive applications. In *SIGCOMM*, 2013.

[25] G. Peng. CDN: content distribution network. *CoRR*, cs.NI/0411069, 2004.

[26] D. M. W. Powers. Applications and explanations of zipf's law. In *Proceedings of the Joint Conferences on New Methods in Language Processing and Computational Natural Language Learning*, NeMLaP3/CoNLL '98, pages 151–160, Stroudsburg, PA, USA, 1998. Association for Computational Linguistics.

[27] J. Shafer, S. Rixner, and A. Cox. The Hadoop distributed filesystem: Balancing portability and performance. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 122–133, March 2010.

[28] H. Shen. Ead: An efficient and adaptive decentralized file replication algorithm in p2p file sharing systems. In *Peer-to-Peer Computing , 2008. P2P '08. Eighth International Conference on*, pages 99–108, Sept 2008.

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[30] The ALICE Collaboration, K. Aamodt, A. A. Quintana, R. Achenbach, S. Acounis, D. Adamová, C. Adler, M. Aggarwal, F. Agnese, G. A. Rinella, et al. The ALICE experiment at the CERN LHC. *Journal of Instrumentation*, 3(08):S08002, 2008.

[31] Q. Wei, B. Veeravalli, and Z. Li. Dynamic replication management for object-based storage system. In *Networking, Architecture and Storage (NAS), 2010 IEEE Fifth International Conference on*, pages 412–419, July 2010.